VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

SHADOW MAPPING: FILTRACE STÍNŮ V OPENGL

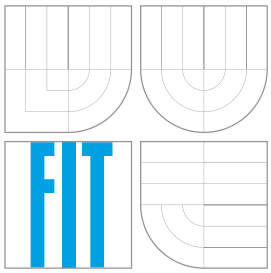
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

ŠIMON MARESZ

BRNO 2014

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

SHADOW MAPPING: FILTRACE STÍNŮ V OPENGL

SHADOW MAPPING: SHADOW FILTERING IN OPENGL

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

ŠIMON MARESZ

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. JOZEF KOBRTEK

BRNO 2014

Abstrakt

Tato práce se zabývá filtrací vykreslovaných stínů ve 3D scéně se zaměřením na kvalitu vykreslených stínů. V rámci této práce byla vytvořena vzorová aplikace implementující vybrané algoritmy, které upravují algoritmus stínových map tak, aby výsledné stíny byly měkčí a reálnější. Program byl vytvořen ve vývojovém prostředí Microsoft Visual Studio 2010 v jazyce C++. Na začátku práce jsou vysvětleny algoritmy pro vykreslení stínů. Závěr práce se zabývá testováním a porovnáním dosažených výsledků.

Abstract

This thesis deals with filtration of shadow rendering in 3D scene with a focus on quality of rendered shadows. As part of this work was to create a sample application that implements the chosen algorithms that modifies the Shadow Mapping algorithm to render softer and more realistic shadows. The program was created in the integrated development environment Microsoft Visual Studio 2010 in C++. At the beginning the algorithms are explained for rendering of shadows. The conclusion is then engaged in testing and comparing of achieved results.

Klíčová slova

OpenGL, Variantní, Exponenciální, Konvoluční Stínové mapy, Percentage-closer filtering, měkké stíny, filtrace

Keywords

OpenGL, Variance, Exponential, Convolution Shadow Mapping, Percentage-closer filtering, soft shadows, filtration

Citace

Šimon Maresz: Shadow Mapping: filtrace stínů v OpenGL, bakalářská práce, Brno, FIT VUT v Brně, 2014

Shadow Mapping: filtrace stínů v OpenGL

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Jozefa Kobrtka, uvedl jsem všechny publikace a literárni prameny, ze kterých jsem čerpal.

.....
Šimon Maresz
21. května 2014

Poděkování

Rád bych poděkoval panu Ing. Jozefu Kobrtkovi jakožto vedoucímu práce za odbornou pomoc, poskytnutí materiálů a rad.

Obsah

Kapitola 1

Úvod

Tato práce se zabývá metodami vykreslování měkkých stínů ve 3D scéně a jejich porovnáním. Pro vykreslování měkkých stínů dnes existuje již mnoho algoritmů, avšak tato práce se bude zabývat jen Shadow mappingem a jeho modifikacemi.

Dnes se nejčastěji používají tři základní algoritmy, a to sice lightmapping, tedy předpočítání stínů do textur objektů; tento algoritmus je ovšem pouze pro statické objekty a nijak nezvyšuje dobu vykreslení scény, algoritmus stínových těles a algoritmus Shadow mapping. Algoritmus již dokáže vypočítat dynamické stíny, ovšem v základní verzi má výrazné nedostatky, jako například vykreslování pouze tvrdých stínů, aliasing.

Tyto nedostatky se snaží odstranit různé modifikace a jimi se budeme zabývat. Pro tuto práci jsme zvolili Percentage-closer filtering, Variance, Exponential a Convolution Shadow mapping.

V jednotlivých kapitolách si rozebereme problémy algoritmu stínových map a jeho implementaci za pomoci knihovny OpenGL, modifikace tohoto algoritmu a v poslední kapitole nalezneme porovnání výsledků jednotlivých modifikací.

Kapitola 2

Přehled algoritmů, jejich modifikací a nedostatků

V této kapitole si popíšeme algoritmus Stínových map v základní formě a jeho nedostatky. Následně budou vysvětleny jednotlivé modifikace tohoto algoritmu řešící tyto nedostatky, a to sice ve formách Percentage-closer filtering, Variance, Exponential a Convolution Shadow mapping. Všechny tyto algoritmy upravují stínové mapy tak, aby výsledný stín byl měkčí a reálnější.

2.1 Shadow mapping

Algoritmus stínových map[7] je proces, díky kterému můžeme vykreslovat stíny ve 3D scéně generovány směrovým světlem. Tento algoritmus se dodnes používá jak ve předem renderovaných scénách, tak ve scénách vykreslovaných v reálném čase. Jelikož tento algoritmus měl několik nedostatků, byl postupem času modifikován. V základní formě má tento algoritmus nízké paměťové i výpočetní nároky.

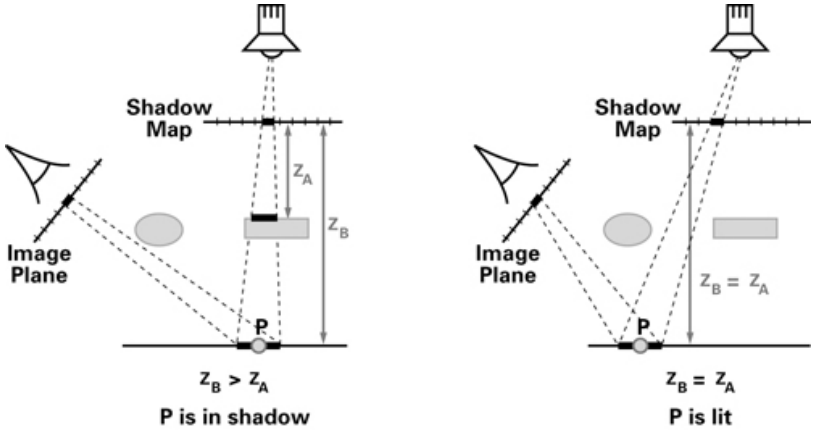
Algoritmus využívá dvou pohledů na scénu, kde první pohled na scénu je veden z kamery uživatele a druhý pohled je veden ze zdroje světla. Poté, při vykreslování scény, jsou jednotlivé pixely testovány na viditelnost ze zdroje světla, pakliže vidět jsou, jsou osvětleny. V opačném případě jsou ve stínu.

Kvůli tomuto porovnáváni je potřeba mít uloženou stínovou mapu, tedy scénu vykreslenou z pohledu světla. Z toho vyplývá fakt, že danou scénu vykreslujeme v každém snímku dvakrát.

1. Renderování stínové mapy.
2. Porovnávání vzdáleností bodů od zdroje světla.
3. Renderování samotné scény se stiny.

2.1.1 Stínová mapa

Jako stínová mapa je použita textura. Do této textury se v každém snímku zaznamenává hloubka scény z pohledu kamery, tedy vzdálenost každého bodu objektu ve viditelném



Obrázek 2.1: Princip algoritmu stínových map[1]

frustru světla. Tato textura je připojena jako příloha hloubky k aktivnímu framebuffer objektu (dále FBO). Tudíž když vykreslujeme do tohoto FBO, automaticky se zaznamená hloubka do textury. Tato textura se poté v druhém cyklu vykreslení použije pro testování viditelnosti. Pro scény se statickým světlem stačí vykreslit tuto texturu jen jednou.

Testování viditelnosti z pohledu světla se provádí ve fragment shaderu. Do fragment shaderu si pošleme souřadnice bodu objektu z pohledu světla. Nově získanými souřadnicemi se poté dotážeme stínové mapy na vzdálenost nejbližšího bodu ke světlu, a porovnáme s vzdáleností aktuálně vykreslovaného bodu. V případě, že je vzdálenost aktualního bodu větší než vzdálenost nejbližšího bodu, vykreslíme jej ve stínu. Pakliže je vzdálenost stejná, vykreslíme jej osvětleně.

2.1.2 Problémy algoritmu

Algoritmus stínových map má několik nedostatků. Jako hlavní si zde popíšeme Shadow acne, neboli nesprávné stínování[2] a aliasing.

- Shadow acne

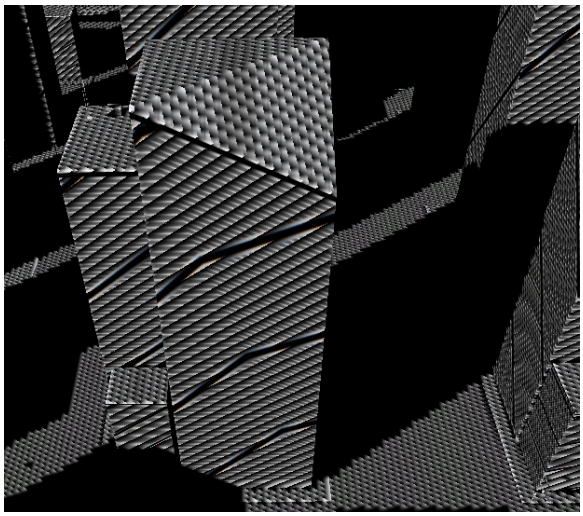
  Shadow acne je problém, kdy jakýkoliv povrch objektu vrhá stín sám na sebe. To je způsobeno nepřesností hloubky stínové mapy, jelikož uložená hloubka v textuře bude vždy méně přesná než hloubka právě vypočteného bodu. Několika sousedícím bodům objektu může odpovídat jeden bod stínové mapy, pote se některé body jevi ve stínu, i když v něm reálně nejsou. Tento problém se nejčastěji řeší pomocí offsetu, tedy odečtením malé hodnoty od vzdálenosti získané z textury. Je ovšem důležité zvolit vhodnou a malou hodnotu tohoto offsetu, v opačném případě dochází ke zkreslení stínů.

- Aliasing

  Aliasing je nedostatek způsobený principem algoritmu stínových map. Projevuje se ostrým zoubkovaným přechodem mezi osvětlenou oblastí a oblastí ve stínu. Tento

[1] Převzato z: http://http.developer.nvidia.com/CgTutorial/cg_tutorial_chapter09.html
[2] incorrect self-shadowing

problém řeši modifikace algoritmu stínových map převážně rozmazáním stínové mapy, nebo mnohonásobným dotazováním stínové mapy.



Obrázek 2.2: Obrázek znázorňuje Shadow acne

2.2 Percentage-closer filtering

Percentage-closer filtering[6] patří mezi jednodušší modifikace algoritmu stínových map. Algoritmus se odlišuje v tom, že místo dotazování aktuálně vykreslovaného bodu se dotazujeme na body v jeho okolí, a poté je hodnota výsledného stínu vydělena počtem testů okolních bodů. Rychlost a kvalita stínů tohoto algoritmu se odvíjí od počtu dotazovaných bodů, platí tedy čím více dotázaných bodů, tím plynulejší přechod mezi stíny a delší doba pro vykreslení snímku.

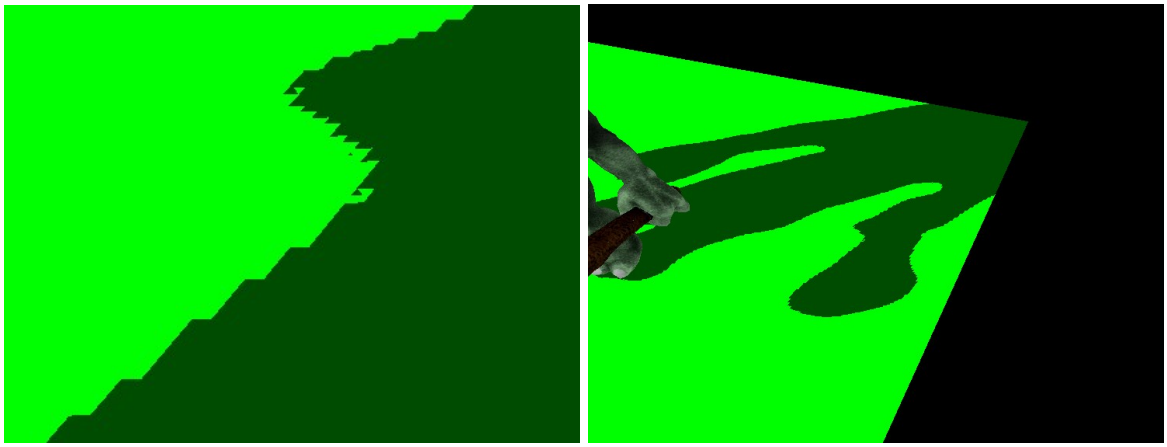
Stínová mapa se při tomto algoritmu nijak nemění, pouze dotazování nad ní je prováděno pro více bodů v okolí.

Obrázek 2.3: Příklad stínové mapy

2.2.1 Problémy algoritmu

Pro plynulé přechody mezi stíny je zapotřebí dotazovat se na velké množství bodů. Pro srovnatelné stíny s Exponenciálními nebo Variantnimi stínovými mapami je potřeba dotazu 8x8 okolních bodů. Dotazování tolika bodů je ovšem náročné, a tím pádem rychlost vykreslení snímku upadá.

Algoritmus je intuitivní a jednoduchý na implementaci. Výhodou tohoto algoritmu je možnost zvolit si strmost přechodu mezi osvětlenou oblastí a oblastí ve stínu. Další výhodou je možnost výběru kvality tohoto přechodu v závislosti na výpočetní době; ne vždy potřebujeme mít nejreálnější stíny.



Obrázek 2.4: Obrázky znázorňují aliasing

2.3 Variance Shadow Mapping

Variance Shadow Mapping[3] se odlišuje tím, že do stínové mapy nezapisují pouze vzdálenost vykreslovaného bodu od světla, ale také různá další data. Tato data zde zastupují rozložení pravděpodobnosti zastínění objektu a jsou použita pro výpočet střední kvadratické odchylky[3], která představuje zastínění okolních bodů.

1. Renderování stínové mapy.
2. Rozmazání stínové mapy v ose x (aplikace Gaussian filtru).
3. Rozmazání stínové mapy v ose y (aplikace Gaussian filtru).
4. Porovnávání vzdáleností bodů od zdroje světla.
5. Renderování samotné scény se stíny.

[3] variance

2.3.1 Stínová mapa

Jelikož Variance Shadow Mapping již využivá dvou složek uložených do stínové mapy, je potřeba stínovou mapu ukládat již jako RGB texturu, a nikoliv jednosložkovou texturu hloubky. Pro vykreslení do takovéto textury jsou již potřeba napsat shadery, které budou schopny ukládat data do jednotlivých složek textury. Variance Shadow Mapping tedy do stínové mapy ukládá mimo hloubky scény také její druhou mocninu. Na takto vytvořenou stínovou mapu lze aplikovat různé filtry, v našem případě aplikujeme Gaussian filtr, díky kterému je stínová mapa rozmazána v obou osách. Takto vytvořená textura je poté testována.

Body jsou poté testovány pomocí Chebyshevovy nerovnosti pro distribuci náhodných proměnných, kde z dat uložených ve stínové mapě vypočteme horní hranici odhadu distribuce zastíněného fragmentu, která představuje pravděpodobnost, že daný fragment je zastíněn.

2.3.2 Prosvítání částečných stínů

Zásadní nedostatek tohoto algoritmu je tzv. prosvítání částečných stínů[4]. Tento jev se projevuje tím, že pokud je objekt zastíněn hierarchií objektů, prosvítají hrany stínů objektů bližších, než je nejvzdálenější objekt. Omezením variance shora lze minimalizovat prosvítání, ovšem za cenu celkového ztmavení, a tím pádem snížením kvality polostínu.

Obrázek 2.5: Obrázek znázorňuje prosvítání polostinů[5]

2.4 Exponential Shadow Mapping

Exponential Shadow Mapping[2] vychází z předchozího algoritmu (VSM). Hlavním rozdílem je použití vhodnější aproximace pro stínovou funkci, díky níž se značně snižuje prosvítání částečných stínů, další výhodou algoritmu je, že do stínové mapy se ukládá pouze vzdálenost bodu od světla.

[4] light-bleeding
[5] Převzato z: http://www.msalvi.org

1. Renderování stínové mapy.
2. Rozmazání stínové mapy v ose x (aplikace Gaussian filtru).
3. Rozmazání stínové mapy v ose y (aplikace Gaussian filtru).
4. Porovnávání vzdáleností bodů od zdroje světla.
5. Renderování samotné scény se stíny.

2.4.1 Stínová mapa

Stínová mapa algoritmu Exponential Shadow Mapping využívá pouze jednu složku, a to sice vzdálenost objektu od zdroje světla. Na tuto vzdálenost poté aplikuje dvoukrokový Gaussian filtr (osa x, poté osa y).

Zastínění fragmentů se počítá obdobně jako u VSM s využitím horního odhadu. Využívá se horniho odhadu vycházejícího z Markovovy nerovnosti.

Díky absenci variance trpí výsledný stín na prosvítání částečných stinů méně než VSM. Další problém nastává v případě, že máme dva objekty, které se dotýkají a jeden vrhá stín na druhý; tento stín se směrem k místu kontaktu zesvětluje.

2.5 Convolution Shadow Mapping

Konvoluční stínové mapy[1] jsou nejsložitějším algoritmem, kterým se v této práci budeme zabývat. Vychází z faktu, že stínovací funkc, která je reprezentována stínovou mapou, lze rozložit pomoci Fourierovy transformace na bázové funkce reprezentované bázovými mapami. Tyto mapy jsou poté filtrovány a opět skládány dohromady 2.1 pro vytvoření měkkého stínu.

1. Renderování stínové mapy.
2. Generování bázových map.
3. Filtrace bázových map.
4. Generování Mipmap bázových map.
5. Renderování samotné scény se stíny (skládání bázových map).

$$f(d,z) \approx \frac{1}{2} + 2\sum_{k=1}^{M} \frac{1}{C_k} cos(C_k * d) * sin(C_k * z) - 2\sum_{k=1}^{M} sin(C_k * d) * cos(C_k * z) \qquad (2.1)$$

2.5.1 Stínová mapa

Stínová mapa je vykreslena klasickým způsobem, ukládá pouze vzdálenost nejbližších osvětlených objektů ke zdroji světla. Tato mapa se poté rozkládá na několik bázových map, které se filtrují a poté sčítají pro vypočtení výsledného stínu.

2.5.2 Bázové mapy

Bázové mapy představují jednotlivé složky stínové funkce. Do těchto map se ukládají tedy sinusovky a cosinusovky hloubky scény. Počet bázových map je kritickým faktorem tohoto algoritmu. Pro dosažení nejlepších výsledků bylo použito 16 sad map, tedy stínová funkce byla rozložena na 16 harmonických funkcí. Každá sada bázových map se skládá ze 4 složek, a to sice: $cos(C_k * d), cos(C_k * z), sin(C_k * d)$ a$sin(C_k * z)$

Jelikož bázové mapy se vykreslují před samotným vykreslením scény, neznáme ještě vzdálenost aktuálně vykreslovaného bodu od zdroje světla; ta se dopočítává při konečném vykreslení. Proto pro každou sadu bázových map ukládáme pouze 2 složky, a to sice sin(Ck z) a cos(Ck z). Při 16 sadách jsme schopni pomocí techniky Multiple Render Target vykreslit všechny mapy do 2 RGBA textur během dvou renderovacích cyklů.

Při vykreslování scény se dotazujeme do každé bázové mapy, získaná data spojíme podle vzorce 2.1, jehož výsledkem je hodnota stínu.

Kapitola 3

Návrh aplikace

Tato kapitola popisuje návrh aplikace, která je vytvořena v rámci této práce. Postupně budou probrány požadavky na aplikaci a návrh jejich řešení. Aplikace je napsaná v jazyce C++ ve vývojovém prostředí Microsoft Visual Studio 2011, shadery jsou psány v jazyce GLSL, který je svou syntaxí podobný jazyku C. Pro práci s grafickou kartou slouží knihovna OpenGL verze 4.3.

3.1 Požadavky na aplikaci

Cílem této práce je porovnat výkon a kvalitu stínů vykreslených vybranými algoritmy. K tomuto účelu je vytvořena aplikace, která bude načítat modely, z těchto modelů vytvářet scény a ty poté vykreslovat se stíny podle jednotlivých algoritmů. Pro objektivní testování a porovnávání jednotlivých algoritmů bude sloužit animace, která bude pohybovat kamerou po zvolené křivce. Jako metrika výkonu je zvolena doba vykreslení jednotlivých snímků animace. Pro pohodlnější práci s aplikací bude implementováno přepínání scén a volně pohyblivá kamera.

Kapitola 4

OpenGL

Tato kapitola popisuje základní pojmy a mechaniky knihovny pro práci s grafickou kartou OpenGL, jelikož v následující kapitole bude popsána implementace algoritmů, ve které budou tyto pojmy použity.

4.1 Shadery

Shader[5] je počítačový program, který slouží k řizení jednotlivých částí programovatelné pipeline. Tyto programy jsou psány speciálním jazykem, a to sice *GLSL*, neboli *OpenGL Shading Language.* Shadery se dělí na několik základních typů, a to sice *Vertex, Tessellation control, Evaluation, Geometry, Fragment a Compute shader*. Mezi těmito shadery si také můžeme posílat data.

V naši aplikaci se využívá pouze *Vertex a Fragment Shader*. Postup programování jednotlivých shaderu se nazývá vykreslovácí řetězec[1]. Tato technika je implementovaná od verze OpenGL verze 2.0, ovšem pro nejlepší výsledky bude použita verze OpenGL verze 4.3.

Druhým postupem vykreslování je fixní řetězec [2]. Tato technika ovšem byla z hlediska vykreslování scény velmi striktní a dosažení různých efektů bylo velmi složité.

Vertex shader je program, který se vykoná nad každým vrcholem modelu scény. Tedy vstupem tohoto programu je vždy vrchol modelu. Dále mohou být do shaderu zaslány tzv. uniformy, neboli proměnné, které budou pro každý shader stejné, např. matice kamery. Výstupem jsou již souřadnice tohoto bodu na zobrazovací ploše. Tyto souřadnice získáme vynásobením tohoto bodu *ViewProjection* maticí kamery. Dále se zde často počítají například normály modelu.

Cílem *fragment shaderu* je výpočet barvy jednotlivých pixelů zobrazovací plochy. Jako vstup je souřadnice jednotlivých bodů, výstupem je barva. Zde se provádí napřiklad texturování, nebo globální osvětlování.

[1] programmable pipeline
[2] fixed pipeline

4.2 Framebuffer objekt

Framebuffer objekt[3] [4] je objekt, který představuje výstupní médium. Pakliže nechceme vykreslovat na obrazovku počítače, můžeme si vytvořit takovýto objekt, kterému nastavíme atributy a přídavky[4], a do něj poté vykreslovat. Jako přídavky můžeme nastavit textury, které můžeme použít k dalšímu vykreslování.

V našem případě budeme využívat Framebuffer objekt převážně k tvorbě stínových map. Jako atributy Framebuffer objektu můžeme nastavit například vykreslování pouze hloubky scény a zakázat vykreslování barevné složky. Takovéto vykreslování je několikanásobně rychlejší.

4.3 Multipler Render Targets

Multiple Render Targets je technika, při které připojíme do vykreslovaného FBO více komponent (`GL_COLOR_ATTACHMENT0`). Díky této techniky jsme schopni v rámci jednoho vykreslovacího cyklu zapsat hodnoty do více komponent najednou. Výstupem fragment shaderu je tedy pole `gl_FragData[]`. Funkcí `glDrawBuffers()` se ještě upřesní seznam bufferů, do kterých se bude vykreslovat.

[3] framebuffer object
[4] attachment

Kapitola 5

Shadow Mapping v OpenGL

V této kapitole budou vysvětleny jednotlivé detaily implementace aplikace. Budou zde popsány důležité datové struktury, které bylo nutné použít pro správnou funkčnost aplikace. Následně použité knihovny pro načítání vstupních souborů (textury, 3D modely), práci s maticemi, zpracovávání vstupů, nakonec zde bude popsána implementace jednotlivých algoritmů pro vykreslování stínů.

5.1 Použité knihovny

Pro tuto práci bylo použito několik knihoven. Tyto knihovny umožňují pracovat s okny, načítat modely a jejich textury, pracovat s maticemi a grafickými kartami.

- Simple Directmedia Layer, tzv. SDL, je multiplatformní knihovna, díky které můžeme jednoduše pracovat se vstupními periferiemi. Tuto knihovnu používáme na vytvoření okna a zpracovávání vstupů uživatele, které potřebujeme pro pohyb kamery.
- OpenGL Extension Wrangler Library Glew je knihovna, která zajišťuje práci s OpenGL. Díky této knihovny můžeme do vytvořeného okna vykreslit naši scénu, pracovat se shadery, vytvořit texturu nejen pro stínovou mapu, ale pro mnoho dalších potřebných věcí a mechanismů.
- Open Asset Import Library. Assimp se používá pro načítání 3D modelů do datové struktury. Knihovna je velmi intuitivní, a tudíž nám usnadňuje práci. Assimp podporuje velkou škálu formátů 3D modelů, od 3ds Max 3DS (.3ds) až po Wavefront Object (.obj).
- Developer's Image Library. DevIl je knihovna pro práci s obrázky. Tuto knihovnu využíváme pro načítání textur. Jelikož Assimp nedokáže načítat textury ze samostatných souborů, je použita tato knihovna. Načítání textur provádí také třída `AssimpModel` pomocí metody `LoadTexFromFile`.
- OpenGL Mathematics. GLM je matematická knihovna pro aplikace zaměřené na práci s GLSL. Knihovna obsahuje funkce pro práci s maticemi. Tato knihovna se převážně využívá pro práci s kamerami, ale také při načítání modelů.

5.2 Základní mechaniky aplikace

Program je tvořen z několika základních kroků. Jako první se inicializují knihovny SDL a glew, které jsou potřebné k vytvoření okna a práci s OpenGL. Poté je vytvořena instance třídy `ShadowMappingApplication,` která zaštiťuje celou aplikaci. Součástí této třídy jsou metody:

- `Initialize()`

  Tato metoda inicializuje další třídy aplikace, například `SceneManager,` která obsahuje seznam scén. Dále se zde nastavují další přepínače knihoven. Tato metoda je jako jediná volaná mimo nekonečnou smyčku.

- `Update()`

  Metoda `Update()` je volaná před vykreslením každého snímku, prochází třídu `InputManager,` ve které jsou registrovány veškeré periferie, jež je potřeba upravovat v každém snímku. Dále se zde provádí kontrola stlačených tlačítek myší, či její pohyb.

- `Draw()`

  Jako poslední v daném snímku se volá metoda `Draw().` Jak už napovídá název, tato funkce volá `glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)` pro vyčištění jednotlivých bufferů. Poté získá ze `SceneManager`u aktivní scénu a zavolá její metodu `Draw().`

Aplikace byla tvořena s cílem co nejvyšší rozšiřitelnosti, jednoduchosti a znovupoužitelnosti. Za tímto účelem bylo také použito několik rozhraní a managerů (scény, kamery, periferie).

5.2.1 Kamera a světlo

Kamera je v aplikaci reprezentovaná jako instance třídy `FirstPersonCamera.` Třída implementuje rozhraní `IInputable.` Rozhraní shromažďuje veškeré instance tříd, jejichž obsah se může měnit při stisku klávesy, nebo pohybu myši. Definuje metody `onKeyDown()` a `onMouseMove().` Takto registrované třídy jsou ukládány do `InputManager`u. Tento manager při stisku klávesy, nebo pohybu myši, prochází seznam registrovaných tříd a provede nad nimi operaci dle akce. Další rozhraní, které implementuje, je `ICamera.` Princip tohoto rozhraní je obdobný jako u předchozího. Shromažďuje veškeré registrované kamery do `CameraManager`u, kde každá kamera musí implementovat metodu `GetName(),` která slouží jakožto hlavní identifikátor kamery. Toto rozhraní je implementováno za účelem možnosti přepínání mezi více kamerami.

- Tvorba kamery. Samotná kamera je vytvořena až při inicializaci scény, a to pomocí konstruktoru. `FirstPersonCamera().` Tento konstruktor vyžaduje tři parametry, a to sice:
  - glm::vec3 position, což je pozice samotné kamery.
  - glm::vec3 lookAt reprezentuje bod, na který kamera směřuje.
  - std::string name, tedy název a jednoznačný identifikátor kamery.

- Aktualizace kamery. Aktualizace kamery se provádí při aktualizaci samotné scény. Třída `MyThirdScene` implementuje rozhraní `IScene` a zároveň `IUpdateableComponent`. Tato rozhraní zajištují provedení metody `Update()` před každým vykreslením snímku. Poté je tedy při aktualizaci scény provedena také aktualizace samotné kamery, tedy metoda `FirstPersonCamera::Update()`.

Podobně jako s kamerou se pracuje také se zdrojem světla pro vykreslení stinové mapy. Pro naši aplikaci jsme se rozhodli pro statický zdroj světla, tudíž již není potřeba upravovat každý snímek. Zdroj světla je také reprezentován maticí. Tato matice je poté také zaslaná do shaderu jako uniform. Jelikož by souřadnice jednotlivých bodů po vynásobení maticí byly homogenní, tedy v intervalu [-1,1], a vzorkování stínové mapy se provádí v intervalu [0,1], je matice světla před odesláním do shaderu vynásobena tzv. bias maticí. Pakliže vynásobíme souřadnice bodu nově vytvořenou maticí, dostaneme již hodnoty v intervalu [0,1].

5.2.2 Animace

Pro testování jednotlivých algoritmů je použita animace. Pro animaci byla vybrána křivka kubického splinu, která je na základě tzv. klíčových snímků[1] aproximovaná. Aproximací křivky provádí funkce `BuildPath`. Výsledkem této funkce jsou 3 pole reprezentující pozici, směr pohledu a vektor směřující nahoru[2]. Animaci reprezentuje třida `CameraAnimation`.

5.2.3 Použité shadery

- Shader pro stínové mapy.

  Tento shader je z trojice nejprimitivnější. Ve *vertex shaderu* se pouze vypočtou výsledné souřadnice bodu z pohledu světla, a to sice vynásobením bodů *ViewProjection* maticí světla. *Fragment shader* je prázdný program. Důvodem je textura, která je nastavena tak, aby do ní byla uložena pouze hloubka dané scény. Výsledkem tohoto shaderu je tedy stínová mapa.

- Shader pro vykreslení výsledné scény.

  Vykreslení scény si ovšem vyžaduje složitější shadery. Jednak z důvodu texturování modelu, jednak z kontroly stínů. Pro porovnávání jednotlivých souřadnic si posíláme, z *vertex shaderu* mimo souřadnic bodu vynásobených *ViewProjection* maticí kamery, také souřadnice vynásobené *ViewProjection* maticí světla. Do *fragment shaderu* se také posílají dvě textury. První je textura daného modelu, která obsahuje jeho barvy druhou texturou je samotná stínová mapa. Porovnání zajišťuje funkce `shadow2DProj()`.

Shader je program, který je potřeba kontrolovat a kompilovat. Shadery jsou obaleny třídou `OGLGraphicDevice`. Tato třída obsahuje dvě instance třídy `IShaderObject`, které načtou zdrojový kód shaderu ze souboru. `OGLGraphicDevice` a poté metodou `LinkAndCreateShaderProgram()` daný program zkompiluje (funkce `glLinkProgram()`). Přepínání mezi právě používanými shader programy se provádí pomoci metody `UseShaderProgram()`.

[1] keyframes
[2] Up vector

5.2.4 Načítání modelu

Načítání modelů se provádí pomocí třídy `AssimpModel`, za pomoci knihovny Assimp. Modely načítáme metodou `LoadFromFile()`, kdy za pomoci importeru je vytvořena datová struktura `aiScene`. Z modelu načítáme po jednotlivých meshích. Jeden mesh je nejmenši programově rozlišitelná část 3D modelu. V rámci meshe si ukládáme:

- Vrcholy. Jsou to jednotlivé vrcholy trojúhelníků; z těchto bodů se vytváří VBO[3], se kterým poté pracují shadery.
- Indexy. Jelikož vykreslování se provádí po trojúhelnících, je potřeba zajistit, aby trojúhelníky byly vykresleny správně, tedy aby bylo zřejmé, které body tvoří jeden trojúhelník. K tomu slouží index buffer, který ukládá pořadí jednotlivých vrcholů. Druhým typem je neindexované renderování. V tomto případě se vykreslují trojúhelníky z tří po sobě jdoucích vrcholů uložených ve VBO.
- Texturovací koordináty. Texturovací koordináty slouží jako souřadnice textury, kdy pomocí těchto souřadnic zjistíme barvu daného bodu. Tyto koordináty jsou uloženy ve stejném vertex bufferu jako vrcholy trojúhelníků.
- Dodatečné informace. Jako další informace si při načítání modelu zjišťujeme míry daných meshů. Tato informace je důležitá pro princip algoritmu. Na základě rozměrů jednotlivých meshů jsou vytvářeny jejich bounding boxy.

Tato data se ukládají do datové struktury `AssimpMesh`. Samotný `AssimpModel` poté obsahuje pole těchto meshů. Data ukládaná v struktuře `AssimpMesh` jsou vertex a index buffer, počet vertexů a indexů, id materiálu daného meshe, název textury, identifikátor a míry meshe, neboli dva vektory obsahující maximální a minimální hodnoty souřadnic. Dále si ukládáme celkový počet takto načtených meshů.

Jelikož při vytváření vertex shaderů jsou do bufferu ukládány dva typy dat (souřadnice vertexu a texturovací koordináty), je potřeba tato data od sebe rozlišit. K tomuto byla vytvořena třída popisující tato data, `GeometryDataDescription`. Tato třída na základě indexu (lokace), velikosti dat, střídy a offsetu oddělí jednotlivé složky bufferu. V samotném shaderu se poté dotazujeme na jednotlivé složky pomoci indexu, neboli lokace.

5.2.5 Načítání textur

Načítání textur zajišťuje třída `AssimpModel`. Přesněji metoda `LoadTexFromFile`, za pomoci knihovny DevIl. Tato metoda načte texturu ze souboru do paměti a vrátí jeji identifikátor. Poté je tato textura načtena do texturovací jednotky. Do shaderu se poté jako uniform pošle identifikátor této texturovací jednotky. Vyhledávání informací v textuře se poté provádí pomoci funkce `texture2D()`, jejíž parametry jsou právě identifikátor texturovací jednotky a texturovací koordináty. Samotné texturování se provádí ve fragment shaderu, kdy pomocí textury se zjistí barva jednotlivých bodů a pakliže je daný bod ve stínu, je tato barva vynásobena hodnotou stínu. Tímto se provede ztmavení barvy.

[3] vertex buffer object

Pro vytvoření stínové mapy je prve vytvořena prázdná textura v paměti (funkce `glGenTextures(), glBindTexture(), glTexImage2D()`). Poté se této textuře nastaví parametry tak, aby se nezapisovala barva, ale pouze hloubka jednotlivých bodů a další doplňující parametry(funkce `glTexParameterf()`). Nakonec se zkontroluje, zda byla textura úspěšně vytvořena, připojí se k texturovací jednotce a vytvoří se nový framebuffer object, který zajistí vykreslení do textury.

Samotné vykreslení stínové mapy do textury se provádí v každém snímku, kdy před vykreslením scény se vykreslí stínová mapa za pomoci shader programu k tomu určenému. Tomuto shaderu je jako uniform poslaná pouze matice světla. Poté, co je vykreslena stínová mapa, vykresluje se samotná scéna, provádí se porovnávání hloubky textury a jednotlivých bodů kamery.

5.3 Implementace algoritmů

V této podkapitole bude v krocích popsána implementace jednotlivých algoritmů. Všechny tyto algoritmy využívají stínových map a tak se v jádře od sebe příliž neliši, rozdíly jsou vesměs pouze u vyhodnocování stínů a pomocných procedur. Algoritmus stínových map se implementujeme převážně ve fragment shaderu.

5.3.1 Percentage-Closer Filtering

Shadery se vytvářejí při inicializaci scény. Tento shader program je specifický tím, že cílový framebuffer je textura, do které se vykreslí vzdálenost jednotlivých bodů od zdroje světla. Jako uniform se do tohoto programu posílá pouze view-projection matice zdroje světla; touto maticí se vynásobí jednotlivé body a poté je do textury uložena na svou pozici vzdálenost bodu od světla. Tato vzdálenost se poté porovnává v shaderu při vykreslování samotné scény.

Textura, do které je uložena hloubka scény, je připojena, jakožto hloubková mapa (komponenta) ke framebuffer objektu, který je cílem vykreslení první části. Také je zakázáno zapisování barev do cílového framebuffer objectu. Toto nastavení několikanásobně urychluje vykreslení stínové mapy.

Porovnávání vzdáleností od zdroje světla. Nyní, když už je vykreslení stínové mapy hotovo, je připojen základní framebuffer(`glBindFramebuffer()`). Provádí se tedy vykreslování do okna aplikace. Pomocí metody `UseShaderProgram()` se nastaví další shader, který bude použit pro vykreslení výsledné scény.

Tomuto programu se jakožto uniformy posílají následující data:

- sm_sample - identifikátor texturovací jednotky, která obsahuje texturu se stínovou mapou.
- tex_sample - identifikátor texturovací jednotky, která obsahuje texturu modelu.
- MVPlight - model-view-projection matice světla.
- MVPcamera - model-view-projection matice kamery.

Ve vertex shaderu se nejdříve každý bod vynásobí jak maticí světla, tak maticí kamery. Tímto dostaneme dva body modelu. Tyto body posíláme dále do fragment shaderu. Ve

fragment shaderu se poté vyhledá barva bodu. Nyní, když je nalezena barva bodu, se provede výpočet stínu. Jelikož je implementován algoritmus Percentage-closer filtering, je vytvořena funkce `lookUp(vec2 offset),` která vyhledá, zda-li bod na offsetu je ve stínu, nebo nikoli. Pro naši implementaci PCF byla zvolena tzv. 8x8 filtrace, kdy je zkoumáno 64 okolních bodů, dále 4x4 a 2x2 filtrace. Díky Percentage-closer filtering se zbavíme ostrých stínů a výsledné stíny jsou měkké.

Porovnává se bod, který byl zaslán z vertex shaderu, tedy souřadnice bodu vynásobené maticí světla spolu s texturou, kdy hlavní ukazatel je vzdálenost jednotlivých bodů od zdroje světla. Tedy souřadnice `z` daného bodu spolu s hodnotou textury stínové mapy na stejných souřadnicích.

Vykreslení stínů ve výsledné scéně. Posledním krokem je vynásobení konstanty stínu s barvou získanou z textury modelu. Výsledkem je tmavší odstín barvy.

Implementaci algoritmu PCF můžete nalézt v souboru `MySecondScene.cpp`

5.3.2 Variance Shadow Mapping

Variance Shadow Mapping je implementována v souboru `MyVSMScene.cpp` a od ostatních implementací se liší tím, že do textury se ukládají dvě složky. Samotná hloubka scény a poté její čtverec. Pro texturu reprezentující stínovou mapu byla použita 32-bitová float RGB textura. Jednotlivá data jsou zapisována do prvních dvou kanálů textury.

Jelikož je textura filtrována 1D Gaussian filtrem, je potřeba mít tyto textury vytvořeny dvě. Do první je uložena hloubka scény a její čtverec. Dále je tato textura filtrována v ose X a uložena do druhé textury, poté filtrace v ose Y a uložena do původní textury. Takto filtrovaná textura se používá pro výpočet stínu.

Pro filtraci textury je potřeba vytvořit čtverec[4] , který slouží jako plátno, přes které se textura uloží. Tento čtverec musí pokrývat celý výhled kamery, jinak by se textura nepromítla správně (tedy homogenní souřadnice v ose X a Y od -1 do 1).

Poté, co byla provedena filtrace stínové mapy, se vykresluje samotná scéna. Rozhodování zastínění fragmentu a výpočet se provádí pomoci Chebyshevova horního odhadu:

Algoritmus 1: ChebyshevUpperBound

```
1: if (distance >moments.x)
2:     return 1.0;
3: float variance = moments.y - (moments.x * moments.x);
4: variance = max(variance,0.00002);
5: float d = distance - moments.x;
6: float p_max = variance / (variance + d * d);
7: return p_max;
```

V připadě, že vzdálenost vykreslovaného fragmentu ke světlu je menší než vzdálenost uložená ve stínové mapě, objekt je osvětlen. V opačném připadě je vypočtena variance, ta je omezená shora funkcí `max()`; toto omezení snižuje nežadoucí prosvítání světla, poté je vypočtena samotná hodnota stínu, kterou je vynásobena barva fragmentu.

Použité shadery pro algoritmus VSM jsou: `shadowVSM.vs, shadowVSM.ps` pro tvorbu dvoukanálové stínové mapy. Pro filtraci textur v jednotlivých osách jsou použity shadery

[4] Screen Aligned Quad

`blurr.vs, blurrX.ps a blurrY.ps.` Poslední sadou shaderů pro vykreslování scény se stíny jsou `basicVSM.vs, basicVSM.ps, quadVSM.vs a quadVSM.ps.`

5.3.3 Exponential Shadow Mapping

Exponential Shadow Mapping je implementován v souboru `MyESMScene.cpp.` Implementace algoritmu je obdobná jako u VSM. Tedy jako první krok je opět vykreslená do textury hloubka scény z pohledu světla. Tato stínová mapa je poté filtrována ve dvou krocích postupně v ose X a ose Y. Jako textura byla zvolena 32 bitová float textura.

Obdobně jako u VSM, i zde se používá dvoukroková filtrace za pomoci Gaussian filtru. Je proto vytvořena druhá textura, do které je uložena stínová mapa filtrovaná v ose X. Opět je potřeba využít čtverce, přes který se textura filtruje a ukládá do FBO.

Posledním krokem algoritmu je výpočet stínu. Ten se provádí pomoci Markovovy nerovnosti.

Shadery použity algoritmem ESM jsou `blurr.vs, blurrX.ps a blurrY.ps, shadowVSM.vs, shadowVSM.ps` pro vykreslení a filtraci stínové mapy, `basicESM.vs, basicESM.ps, quadESM.vs a quadESM.ps` pro vykreslení scény se stíny.

5.3.4 Convolution Shadow Mapping

Convolution Shadow Mapping je již složitější na implementaci. Jako první krok je vykreslena stínová mapa, do které se ukládá pouze hloubka scény z pohledu světla. Pro tuto mapu je vytvořena 32 bitová float textura. Takto vytvořená stínová mapa se již nijak neupravuje.

Dalším krokem je tvorba bázových textur. Bázové textury tvoříme pomocí techniky zvané Multi Render Targets. Díky tomu můžeme vytvořit v jednom vykreslovacím cyklu až 4 bázové textury po 4 kanálech. Tedy všech 32 bázových map můžeme vytvořit ve dvou vykreslovacích cyklech. Do bázových map ukládáme $sin(C_k * z)$ a $cos(C_k * z)$, kde C_k je koeficient báze Fourierovy transformace a z je hloubka fragmentu uloženého ve stínové mapě. Opět je vše renderováno za pomocí čtverce, přes který se textura uloží do cílového FBO.

Takto vykresleným bázovým mapám jsou poté generovány Mipmapy automatickým nástrojem grafických karet `glGenerateMipmap(GL_TEXTURE_2D)` . Dále jsou textury filtrovány pomocí dvoukrokového 5x5 Gaussian filtru.

V posledním kroku se vypočtou $sin(C_k * d)$ a $cos(C_k * d)$, kde d je hloubka aktuálně vykreslovaného fragmentu a sečtou se s hodnotami bázových mapy za pomoci vah. Z takovéhoto součtu poté vznikne výsledný stín.

Pro CSM byly napsány shadery pro vytvoření jednokanálové stínové mapy `shadowCSM.ps, shadowCSM.vs.` Shader pro vytvoření bázových funkci `basisGenerator.ps, basisGenerator.vs,` shader pro filtraci bázových funkcí `basisBlurr.vs, basisBlurrX.ps, basisBlurrY.ps`

a share pro výpočet scény se stíny `basicCSM.vs, basicCSM.ps, quadCSM.vs, quadCSM.ps`.

Algoritmus 2: CalculateShadow z quadCSM.ps

```
1: float sum0, sum1;
2: sum0 = 0.0;
3: sum1 = 0.0;
4: vec4 sin_val, cos_val, Ck;
5: sin_val = texture2D(fsin1, upSC.xy);//sum0
6: cos_val = texture2D(fcos1, upSC.xy);//sum1
7: Ck = Ck_v(1);
8: sum0 += cos(Ck.x * z) / Ck.x * sin_val.x*0.3;
9: sum1 += sin(Ck.x * z) / Ck.x * cos_val.x*0.3;
10: sum0 += cos(Ck.y * z) / Ck.y * sin_val.y*0.28;
11: sum1 += sin(Ck.y * z) / Ck.y * cos_val.y*0.28;
12: sum0 += cos(Ck.z * z) / Ck.z * sin_val.z*0.25;
13: sum1 += sin(Ck.z * z) / Ck.z * cos_val.z*0.25;
14: sum0 += cos(Ck.w * z) / Ck.w * sin_val.w*0.22;
15: sum1 += sin(Ck.w * z) / Ck.w * cos_val.w*0.22;
16: sin_val = texture2D(fsin2, upSC.xy);//sum0
17: cos_val = texture2D(fcos2, upSC.xy);//sum1
18: ...
19: Ck = Ck_v(13);
20: sum0 += cos(Ck.x * z) / Ck.x * sin_val.x*0.5;
21: sum1 += sin(Ck.x * z) / Ck.x * cos_val.x*0.5;
22: sum0 += cos(Ck.y * z) / Ck.y * sin_val.y*0.5;
23: sum1 += sin(Ck.y * z) / Ck.y * cos_val.y*0.5;
24: sum0 += cos(Ck.z * z) / Ck.z * sin_val.z*0.5;
25: sum1 += sin(Ck.z * z) / Ck.z * cos_val.z*0.5;
26: sum0 += cos(Ck.w * z) / Ck.w * sin_val.w*0.5;
27: sum1 += sin(Ck.w * z) / Ck.w * cos_val.w*0.5;
28: float rec = 0.5+2.0*(sum0-sum1);
29: rec = clamp(2.0*rec,0.0,1.0);
30: return vec4(rec,rec,rec,rec);
```

Takto jsou vypočteny stíny pomocí CSM, kde textury `fsinx` a `fcosx` jsou bázovými texturami, C_k je koeficient báze Fourierovy transformace, z je hloubka právě vykreslovaného fragmentu. Takto jsou akumulovány hodnoty bázových textur, které jsou poté sečteny dle vzorce.

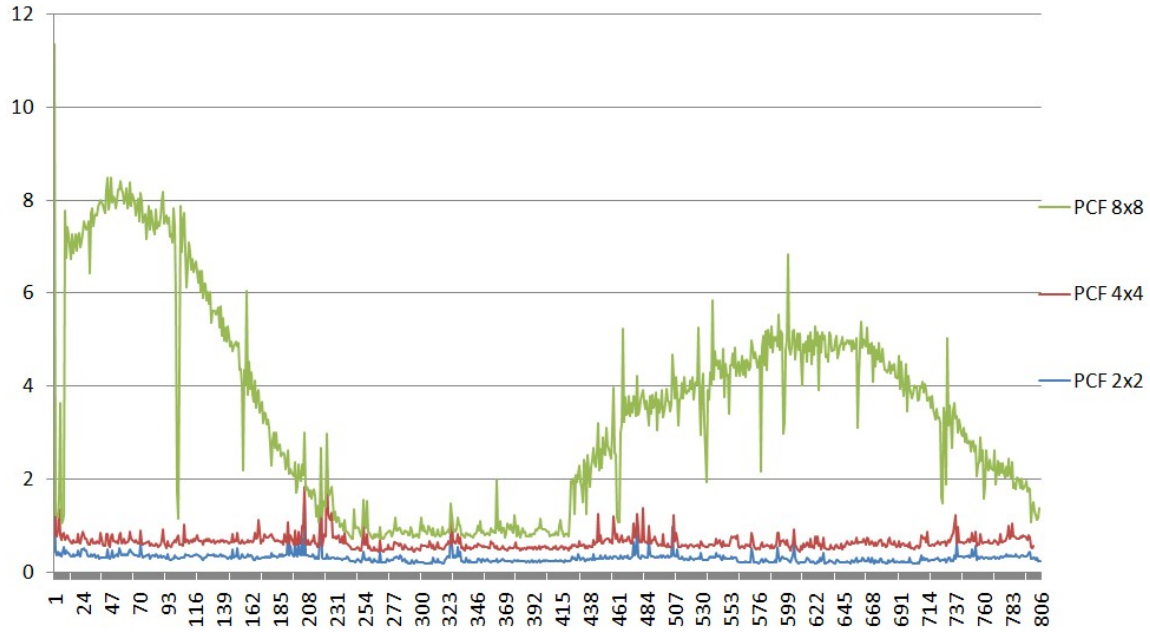
Kapitola 6

Porovnání výkonu a kvality jednotlivých metod

V této kapitole budou jednotlivé algoritmy zhodnoceny. Pro každý algoritmus bude vynesen graf znázorňující dobu vykreslování jednotlivých snímků animace. Obrázky s detailem hrany stínu jsou v příloze. Grafy budou porovnávat daný algoritmus vždy proti PCF v provedení 8x8. Veškeré stinové mapy byly v rozlišení 800x600.

6.1 Percentage-Closer Filtering

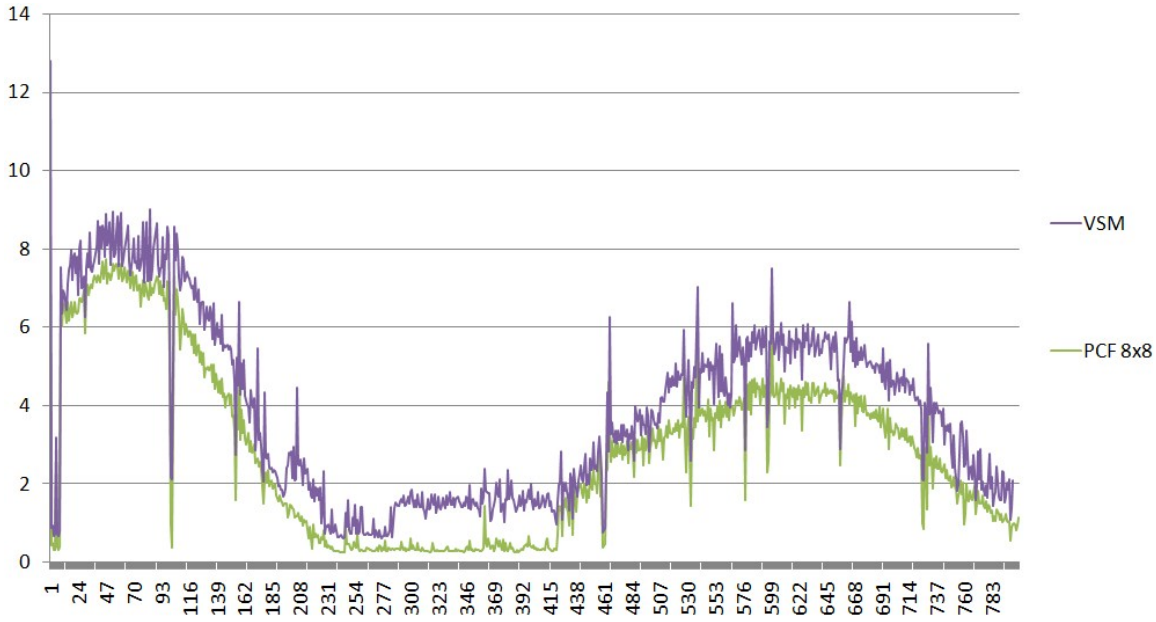
Percentage-Closer Filtering byl implementován ve třech provedeních. Výhodou tohoto algoritmu je možnost dynamicky měnit počet vzorků z okolních bodů. Implementovány byly verze 8x8, 4x4 a 2x2.



Obrázek 6.1: Graf PCF

6.2 Variance Shadow Mapping

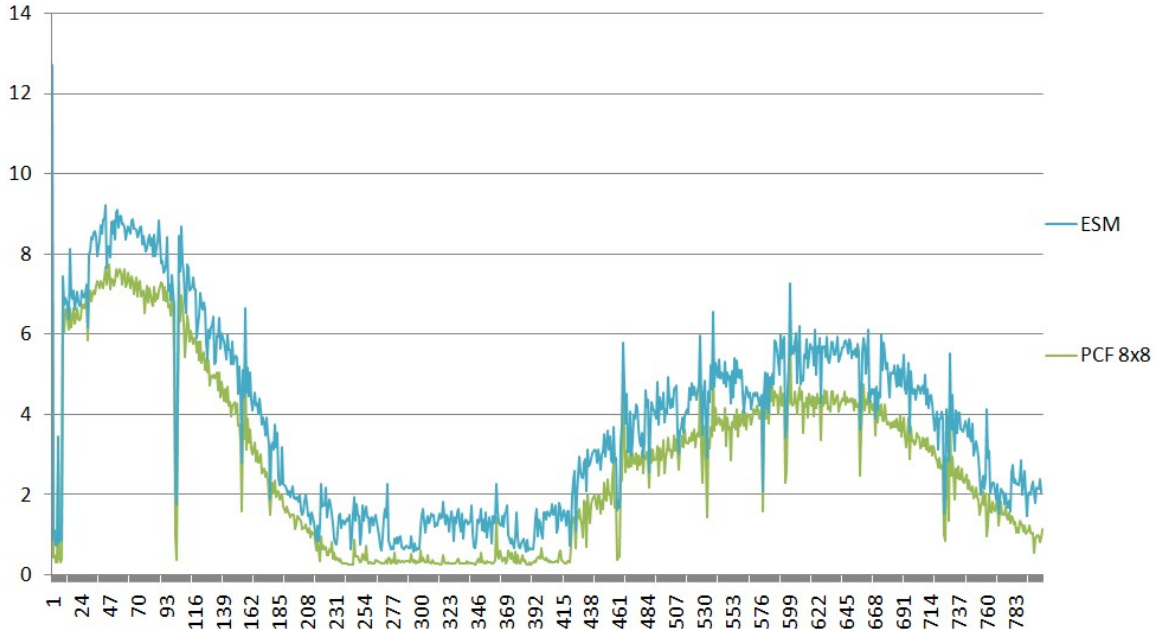
Variance Shadow Mapping má poměrně nízké výpočetní nároky, kvalita stínů je srovnatelná s VSM, avšak značně trpí prosvítáním částečných stínů, proto doporučuji spíše použití ESM.



Obrázek 6.2: Graf VSM - PCF8x8

6.3 Exponential Shadow Mapping

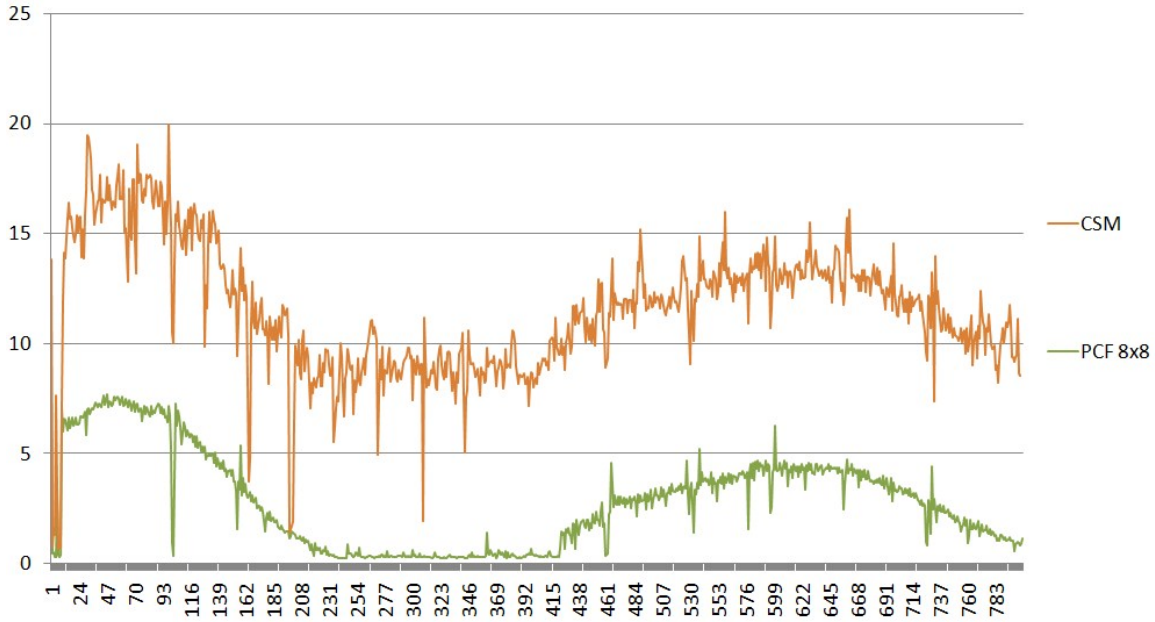
Exponential Shadow Mapping je kvalitou i výkonem srovnatelný s VSM, navíc trpí méně na prosvítání částečných stínů. Algoritmus je jednodušší na nastudování a implementaci.



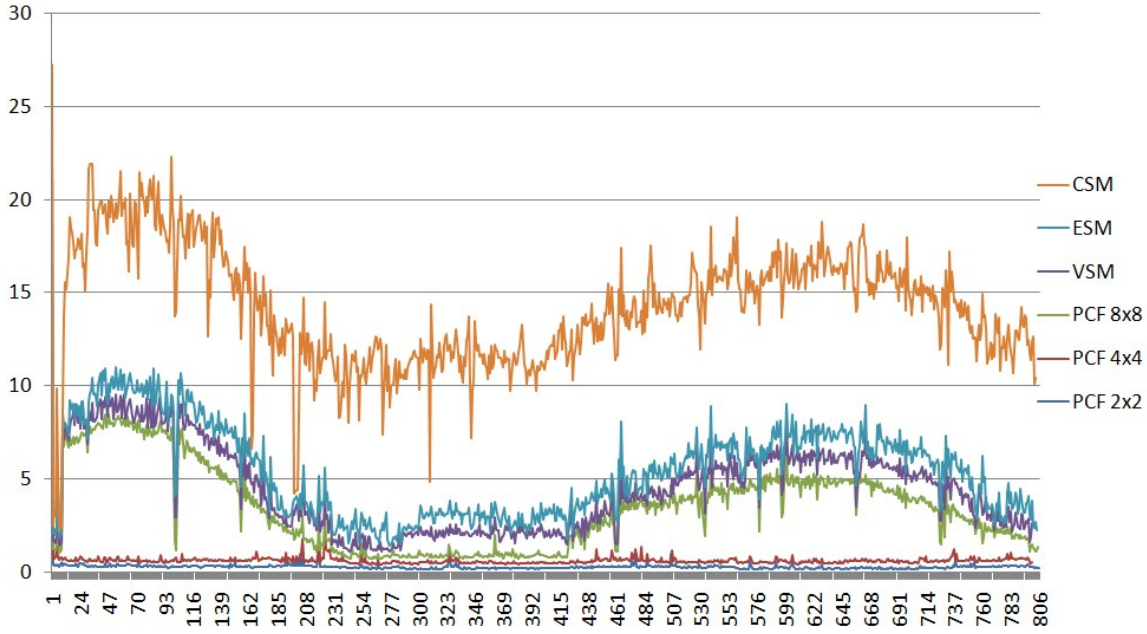
Obrázek 6.3: Graf ESM - PCF8x8

6.4 Convolution Shadow Mapping

Algoritmus Convolution Shadow Mapping dosahuje nejkvalitnějších stínů, avšak je výpočetně poměrně náročný. Algoritmus je celkově složitější pochopit a implementovat. V článku navíc není zmíněno použití vah pro jednotlivé bázové funkce, bez kterých algoritmus nefunguje správně.



Obrázek 6.4: Graf CSM - PCF8x8



Obrázek 6.5: Graf všech algoritmů

Kapitola 7

Závěr

Z výsledků v kapitole 6 můžeme konstatovat, že každý algoritmus nabízí jinou náročnost a kvalitu stínů, tudíž volba správného algoritmu by se měla vždy odvíjet od toho, k čemu daná aplikace bude sloužit a jaké bude mít požadavky na kvalitu stínů.

Nejreálnější stíny jsou vykreslovány algoritmem Convolution Shadow Mapping, avšak tento algoritmus je poměrně náročný na implementaci a výpočet. Exponential Shadow Mapping, který principielně vychází z Variance Shadow Mapping, je kvalitativně na úrovni CSM, avšak v době výpočtu jej předčí. Také prosvítání světla je oproti VSM mnohem nižší, tudíž pro aplikace vykreslované v reálném čase se může jevit jako algoritmus s nejlepším poměrem kvalita/rychlost.

Mezi přednosti algoritmu Percentage-Closer Filtering patří možnost jeho škálování, tedy vykreslování stínů různé kvality, ovšem s rostoucí kvalitou těchto stínů kvadraticky roste počet operací čtení z textury, který je poměrně náročný. Z toho můžeme usoudit, že možnost použití klasického PCF pro kvalitní stíny odpadá.

Jako pokračování této práce by bylo možné přidat další algoritmy a jejich modifikace, jako například lightmapping, nebo právě algoritmus stínových těles, díky kterému by tato práce pokryla i část všesměrových světel. Další možností je vytvořit robustní algoritmus výpočtu stínů, který kombinuje více předchozích algoritmů k vytvoření efektivního vykreslování stínů.

Dalším odvětvím, kterým by se tato práce mohla do budoucnosti zabývat, je optimalizace vykreslování algoritmů stínových map[2] s ohledem na rychlost vykreslování.

Literatura

[1] Annen, T., Mertens, T., Bekaert, P. et al. Convolution Shadow Maps. In *Rendering Techniques 2007: Eurographics Symposium on Rendering*. Grenoble, France: Eurographics, 2007. S. 51–60. ISBN 978-3-905673-52-4.

[2] Annen, T., Mertens, T., Seidel, H.-P. et al. Exponential Shadow Maps. In *GI '08: Proceedings of graphics interface 2008*. Toronto, Ont., Canada: Canadian Information Processing Society, 2008. S. 155–161. ISBN 978-1-56881-423-0.

[3] Donnelly, W. a Lauritzen, A. Variance Shadow Maps. In *Proceedings of the 2006 Symposium on Interactive 3D Graphics and Games*. New York, NY, USA: ACM, 2006. S. 161–165. I3D '06. Dostupné na: <http://doi.acm.org/10.1145/1111411.1111440>. ISBN 1-59593-295-X.

[4] OpenGL. *Framebuffer* [online]. Poslední změna 29. srpna 2013 [cit. 21. května 2014]. Dostupné na: <http://www.opengl.org/wiki/Framebufferr>.

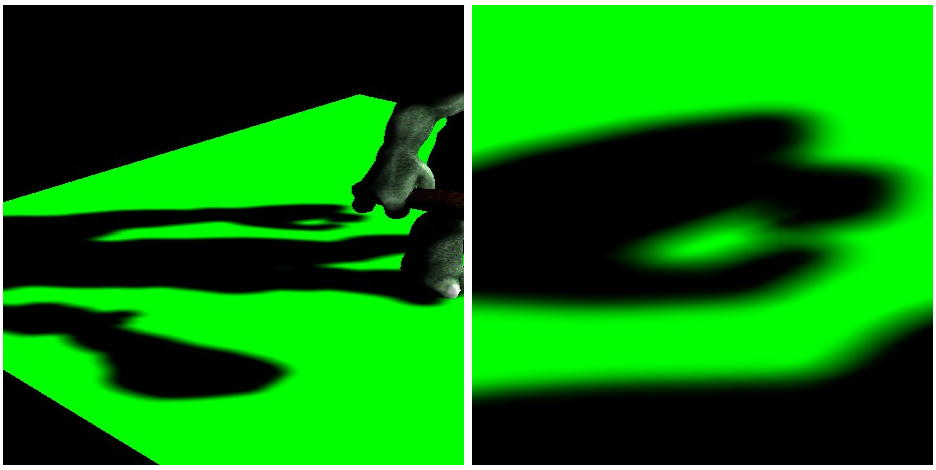
[5] OpenGL. *Shaders* [online]. Poslední změna 31. srpna 2013 [cit. 21. května 2014]. Dostupné na: <http://www.opengl.org/wiki/Shader>.

[6] Reeves, W. T., Salesin, D. H. a Cook, R. L. Rendering Antialiased Shadows with Depth Maps. In *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*. New York, NY, USA: ACM, 1987. S. 283–291. SIGGRAPH '87. Dostupné na: <http://doi.acm.org/10.1145/37401.37435>. ISBN 0-89791-227-6.

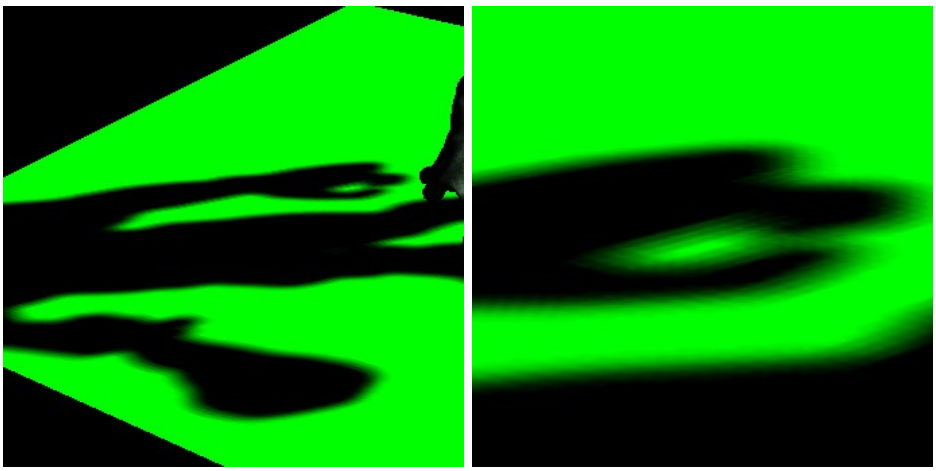
[7] Williams, L. Casting Curved Shadows on Curved Surfaces. In *Proceedings of the 5th Annual Conference on Computer Graphics and Interactive Techniques*. New York, NY, USA: ACM, 1978. S. 270–274. SIGGRAPH '78. Dostupné na: <http://doi.acm.org/10.1145/800248.807402>.

Příloha A

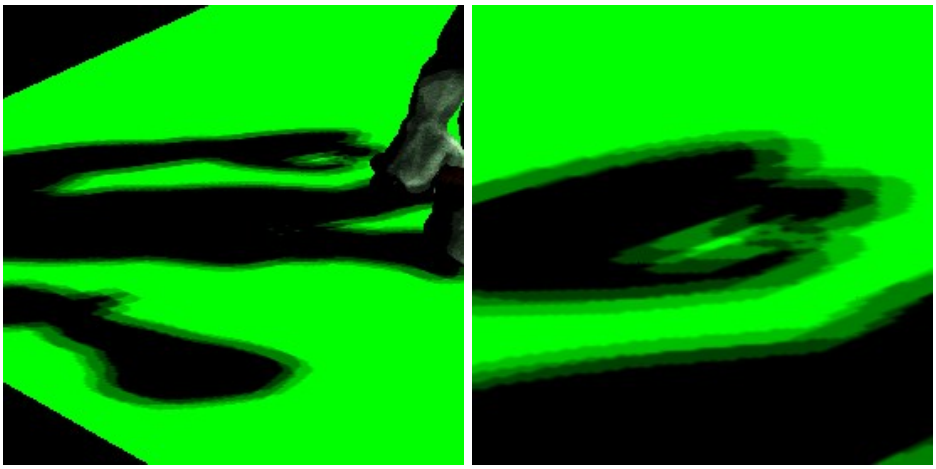
Obrázky stínů



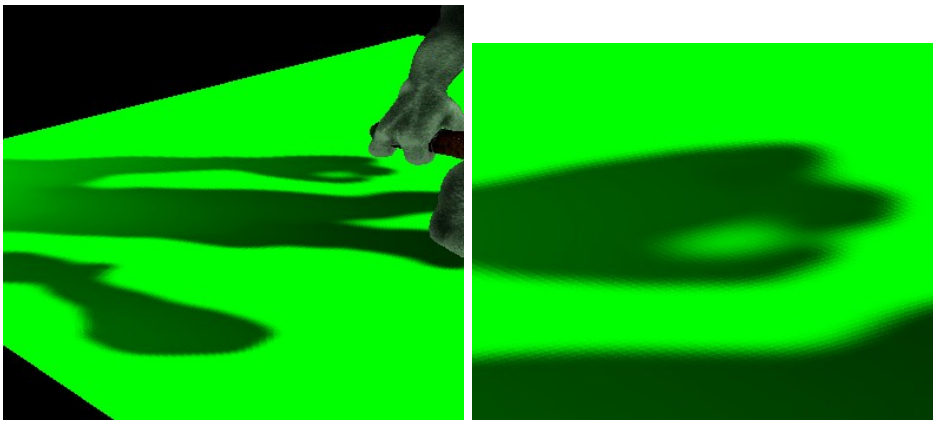
Obrázek A.1: Algoritmus PCF 8x8



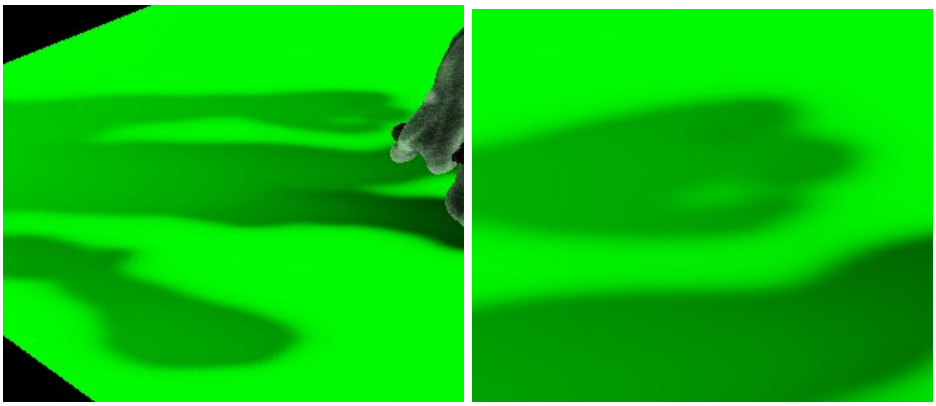
Obrázek A.2: Algoritmus PCF 4x4



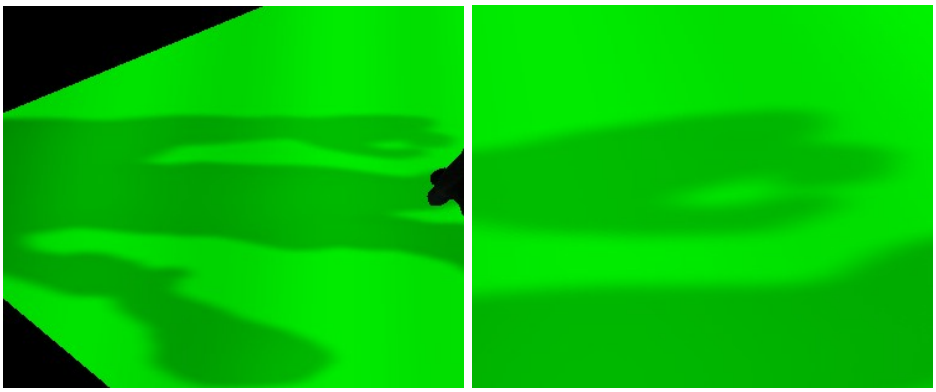
Obrázek A.3: Algoritmus PCF 2x2



Obrázek A.4: Algoritmus VSM



Obrázek A.5: Algoritmus ESM



Obrázek A.6: Algoritmus CSM